MOOCHO Reference Manual

MOOCHO Version 1.0 in Trilinos 7.0

Generated by Doxygen 1.4.2

Tue Sep 26 12:10:22 2006

Contents

1 MOOCHO: Multi-functional Object-Oriented arCHitecture for Optimization

WARNING! This documentation is currently under active construction!

1.1 Outline

- Introduction
- MOOCHO Overview Document
- Hyper-linked HTML version of this Document
- MOOCHO Quickstart
  - Configuring, Building, and Installing MOOCHO
  - Installed Optimization Examples
    - Examples of General Serial NLPs with Explicit Jacobian Entries
    - Examples of Simulation-Constrained NLPs using Thyra
  - Running MOOCHO to Solve Optimization Problems
- Representing Nonlinear Programs for MOOCHO to Solve
  - Representing General Serial NLPs with Explicit Jacobian Entries
  - Representing Simulation-Constrained Parallel NLPs through Thyra
- Other Trilinos Packages on which MOOCHO Directly Depends
- Configuration of the MOOCHO Package
- Individual MOOCHO doxygen collections
- Browse all of MOOCHO as a single doxygen collection
- Links to other documentation collections

1.2 Introduction

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems (i.e. nonlinear programs) using reduced-space successive quadratic programming (SQP) methods. The most general form of the optimization problem that can be solved is:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & c(x) = 0 \\ & x_L \leq x \leq x_U \end{array}$$

where $x \in \Re^n$ the vector of optimization variables, $f(x) \in \Re^n \rightarrow \Re$ is the nonlinear scalar objective function, $c(x) = 0$ (where $c(x) \in \Re^n \rightarrow \Re^m$) are the nonlinear constraints, and x_L and x_U are the upper and lower bounds on the variables. The current algorithms in MOOCHO are well suited to solving optimization problems with massive numbers of unknown variables and equations but few so-called degrees of optimization freedom (i.e. the degrees of freedom = the number of variables minus the number of equality constraints = $n - m$). Various line-search based globalization methods are available, including exact penalty functions and a form of the filter method. Many of the algorithms in MOOCHO are provably locally and globally convergent for a wide class of problems in theory but in practice the behavior and the performance of the algorithms varies greatly from problem to problem.

MOOCHO was initially developed to solve general sparse optimization problems where there is no clear distinction between state variables and optimization parameters. For these types of problems a serial sparse direct solver must be used (i.e. using MA28) to find a square basis that is needed for the variable reduction decompositions that are current supported.

More recently, MOOCHO has been interfaced through `Thyra` and the `Thyra::ModelEvaluator` interface to address very large-scale, massively parallel, simulation-constrained optimization problems that take the form:

$$\begin{array}{ll} \text{minimize} & f(x_D, x_I) \\ \text{subject to} & c(x_D, x_I) = 0 \\ & x_{D,L} \leq x_D \leq x_{D,U} \\ & x_{I,L} \leq x_I \leq x_{I,U} \end{array}$$

where $x_D \in \Re^m$ are the "dependent" state variables, $x_I \in \Re^{n-m}$ are the "independent" optimization parameters and $c(x_D, x_I) = 0$ are the discrete nonlinear state simulation equations. Here the state Jacobian $\frac{\partial c}{\partial x_D}$ must be square and nonsingular and the partitioning of $x = \left[\begin{array}{cc} x_D^T & x_I^T \end{array} \right]^T$ into state variables x_D and optimization variables x_I must be known *a priori* and this partitioning can not change during a solve. Warning, the `Thyra::ModelEvaluator` interface uses a overlapping and inconsistent set set of names for the variables and the problem

functions. All of the functionality needed for MOOCHO to solve a simulation-constrained optimization problem can be specified through sub-classing the `Thyra::ModelEvaluator` interface, and related `Thyra` interfaces. Epetra-based applications can instead implement the `EpetraExt::ModelEvaluator` interface and never need to work with Thyra directly except in trivial and transparent ways.

For simulation-constrained optimization problems, MOOCHO can utilize the full power of the massively parallel iterative linear solvers and preconditioners available in Trilinos through Thyra through the `Stratimikos` package by just flipping a few switches in a parameter list. These include all of the direct solves in Amesos, the preconditioners in Ifpack and ML, and the iterative Krylov solvers in AztecOO and Belos (which is not being released but is available in the development version of Trilinos). For small to moderate numbers of optimization parameters, the only bottleneck to parallel scalability is the linear solver used to solve linear systems involving the state Jacobian $\frac{\partial c}{\partial x_D}$. The reduced-space SQP algorithms in MOOCHO itself exhibit extremely good parallel scalability. The parallel scalability of the linear solvers is controlled by the simulation application and the Trilinos linear solvers and preconditioners themselves. Typically, the parallel scalability is limited by the preconditioners as the problem is partitioned to more and more processes.

MOOCHO also includes a minimally invasive mode for reduced-space SQP where the simulator application only needs to compute the objective and constraint functions $f(x_D, x_I) \in \Re^n \rightarrow \Re$ and $c(x_D, x_I) \in \Re^n \rightarrow \Re^m$ and solve only forward linear systems involving $\frac{\partial c}{\partial x_D}$. All other derivatives can be approximated with directional finite differences but any exact derivatives that can be computed by the application are happily accepted and fully utilized by MOOCHO through the `Thyra::ModelEvaluator` interface.

1.3 MOOCHO Overview Document

A more detailed mathematical overview of nonlinear programming and the algorithms that MOOCHO implements are described in the document `An Overview of MOOCHO`.

1.4 Hyper-linked HTML version of this Document

The doxygen-generated hyper-linked version of his document can be found at the Trilinos website under the link to MOOCHO.

1.5 MOOCHO Quickstart

In order to get started using MOOCHO to solve your NLPs you must first build MOOCHO as part of Trilinos and install it. When MOOCHO is installed with

Trilinos, several complete examples are also installed that show how to define NLPs, compile and link against the installed headers and libraries, and how to run the MOOCHO solvers.

Below, we briefly describe Configuring, Building, and Installing MOOCHO, Running MOOCHO to Solve Optimization Problems, and accessing the Installed Optimization Examples.

1.5.1 Configuring, Building, and Installing MOOCHO

Complete details on the configuration, building, and installation of Trilinos are described in the `Trilinos Users Guide`. However, here we give a quick overview of one such installation.

Here we describe the configuration, build, and installation process for a directory structure that looks like:

```
$TRILINOS_BASE_DIR
 |
 |-- Trilinos
 |
  -- BUILDS
      |
       -- DEBUG
```

where `$TRILINOS_BASE_DIR` is some base directory such as `TRILINOS_BASE_DIR=$HOME/Trilinos.base`. However, in general, the build directory (show as `$TRILINOS_BASE_DIR/BUILDS/DEBUG` above) can be any directory you want but should not be the same as the base directory for Trilinos. In the most general case, we will assume that `$TRILINOS_BUILD_DIR` is the base build directory; in this section, we assume that `TRILINOS_BUILD_DIR=$TRILINOS_BASE_DIR/BUILDS/DEBUG`. The Trilinos Users Guide might still recommend that you create the build directory from within the main Trilinos source directory tree (i.e. `Trilinos/DEBUG`) but we recommend against this practice and the build system supports the more general case described here just as well.

Here are the steps needed to configure, build, and install MOOCHO along with the rest of Trilinos:

1. **Obtain a source tree for Trilinos**

   Once you have created the base directory `$TRILINOS_BASE_DIR` you need to get a copy of the Trilinos source.

   If you have CVS access you can obtain the version of the day through the main development trunk or can check out a specific tagged release. For example, to obtain the version of the day you would perform:

```
cd $TRILINOS_BASE_DIR
cvs -d :ext:userid@software.sandia.gov:/space/CVS co Trilinos
```

where `userid` is your user ID on the CVS server. For further details on working with CVS access to Trilinos, see the Trilinos Developers Guide.

If you do not have CVS access you can obtain a tar ball for a release of Trilinos from the Trilinos Releases Download Page. Once you have the tar ball, you can expand it into the directory `$TRILINOS_BASE_DIR` as follows:

```
cd $TRILINOS_BASE_DIR
tar -xzvf ~/Trilnos-7.0.x.tar.gz
```

where `7.0.x` is some minor release number of Trilinos; hopefully the most current version.

2. **Create the build base directory**

   After you have a copy of the Trilinos source tree in `$TRILINOS_BASE_DIR/Trilinos`, you need to create the base build directory. Here, we assume that you will create the build directory `$TRILINOS_BASE_DIR/BUILDS/DEBUG` as follows:

   ```
   cd $TRILINOS_BASE_DIR
   mkdir BUILDS
   mkdir BUILDS/DEBUG
   ```

3. **Create a configuration script**

   Once you have the Trilinos source code and have created a base build directory, you need to create a configuration script for Trilinos. By far the hardest part of building and installing Trilinos is figuring out how to write the configuration script that will work for the system that you are on and includes the packages and extra options that you need. The best place to find example configure scripts that at least have a chance of being correct on specific systems is to look at Trilinos test harness scripts in the directory:

   ```
   Trilinos/commonTools/test/harness/invoke-configure
   ```

   Older scripts that have worked on a wider variety of systems in the past for different sets of packages can be found in the directory:

   ```
   Trilinos/sampleScripts
   ```

Warning! The scripts in `Trilinos/sampleScripts` are likely to be currently broken for even the same systems for which they where developed. These scripts really only provide ideas for different combinations of options to try to get a configure script to work on your system.

Below is perhaps one of the simplest configure scripts that might get Trilinos to build on a GCC/Linux based platform with MOOCHO support correctly and with enough capability to be useful for initial development purposes:

```
$TRILINOS_BASE_DIR/Trilinos/configure \
--prefix=$TRILINOS_INSTALL_DIR \
--with-gnumake \
--enable-export-makefiles \
--with-cflags="-g -O0 -ansi -Wall" \
--with-cxxflags="-g -O0 -ansi -Wall -pedantic" \
--enable-teuchos-extended --enable-teuchos-debug --enable-teuchos-abc \
--enable-thyra \
--enable-epetraext \
--enable-stratimikos \
--enable-moocho
```

A word of caution is in order about the above simple configure script; The above script assumes that certain packages will be turned on by default (such as Epetra, Amesos, AztecOO, Ifpack, and ML) and other will be turned on automatically by the presence of other enables. While this should work correctly for many different possible combinations of enables and disables, there are many configurations that will not work just due to faults in logic and inadequate testing of all of the possible options. When in doubt, be explicit about what you enable and be weary about selectively disabling certain packages and subpackages.

Below is an example of a more compliicated configure script that might be used to configure Trilinos with MOOCHO support and a Linux system with gcc with more capabilities based on some third-party libraries (but the script might not actually work on any actual computer on Earth):

```
$TRILINOS_BASE_DIR/Trilinos/configure \
--prefix=$TRILINOS_INSTALL_DIR \
--with-install="/usr/bin/install -p" \
--with-gnumake \
--enable-export-makefiles \
--with-cflags="-g -O0 -ansi -Wall" \
--with-cxxflags="-g -O0 -ansi -Wall -ftrapv -pedantic -Wconversion" \
--enable-mpi --with-mpi-compilers \
--with-incdirs="-I${HOME}/include" \
--with-ldflags="-L${HOME}/lib/LINUX_MPI" \
--with-libs="-ldscpack -lumfpack -lamd -lparmetis-3.1 -lmetis-4.0 -lskit" \
--with-blas=-lblas \
--with-lapack=-lapack \
--with-flibs="-lg2c" \
--disable-default-packages \
--enable-teuchos --enable-teuchos-extended --disable-teuchos-complex \
```

```
  --enable-teuchos-abc --enable-teuchos-debug \
--enable-thyra \
--enable-epetra \
--enable-triutils \
--enable-epetraext \
--enable-amesos --enable-amesos-umfpack --enable-amesos-dscpack \
--enable-aztecoo \
--enable-ifpack --enable-ifpack-metis --enable-ifpack-sparskit \
--enable-ml --with-ml_metis --with-ml_parmetis3x \
--enable-stratimikos \
--enable-moocho
```

The above script is almost completely platform dependent in most cases, except for everything below `-disable-default-packages` for enable options for individual packages. A few points about the above configure script are worth mentioning. First, some of the package enable options such as `-enable-epetra` should be unnecessary once other options such as `-enable-epetraext` are included but to be safe it is a good idea to be explicit about what packages to build in case the default built-in top-level configure logic does not handle the dependencies correctly. Second, it is a good idea to include the options `-enable-teuchos-debug` and `-enable-teuchos-abc` when you first start working with Trilinos to help catch coding errors on your part (and perhaps on the part of Trilinos developers). Third, the above script shows enabled support for several third-party libraries such as UMFPACK, DSCPACK, SparseKit, and Metis. You are responsible for installing these third party libraries yourself if you want the extra capabilities that they enable. Otherwise, to get started, a simpler script, such as shown above, can be used to get started with Trilinos/MOOCHO.

As a final step, you can copy the contents of the configure invocation command (examples shown above) into a script called `do-configure` and make the script executable which is assumed below.

4. **Configure, build, and install Trilinos**

   Once you have a configure script, you can try to configure and build Trilinos as follows:

   ```
   cd $TRILINOS_BUILD_DIR
   ./do-configure
   make
   make install
   ```

   If a problem does occur, it usually occurs during configuration. Often trial and error is required to get the configuration to complete successfully.

   Once the Trilinos build completes (which can take hours on a slower machine if a lot of packages are enabled) you should test Trilinos using something like:

```
make runtest-mpi TRILINOS_MPI_GO="mpirun -np "
```

If MPI is not enabled, you run run the serial test suite as:

```
make runtest-serial
```

Some of the tests in the test suite may fail if you have not enabled everything and this is okay. Once you feel confident that the build has completed correctly, you can install Trilinos as follows:

```
make install
```

If everything goes smoothly, then Trilinos will be installed with the following directory structure:

```
$TRILINOS_INSTALL_DIR
 |
 |-- examples
 |
 |-- include
 |
 |-- libs
 |
  -- tools
```

Once the install competes, you can move on to building and running the installed MOOCHO examples as described in the next section.

1.5.2 Installed Optimization Examples

When the configure option `-enable-export-makefiles` is included, a set of examples are installed in the directory specified by `-prefix=$TRILINOS_INSTALL_DIR` and the directory structure will look something like:

```
$TRILINOS_INSTALL_DIR
 |
 |-- examples
 |    |
 |     -- moocho
 |         |
 |         |-- NLPWBCounterExample
 |         |
 |         |-- ExampleNLPBanded
 |         |
```

```
|         |-- thyra
|              |
|              |-- NLPThyraEpetraModelEval4DOpt
|              |
|               -- NLPThyraEpetraAdvDiffReactOpt
|
 -- tools
     |
      -- moocho
```

Note that the directory `$TRILINOS_INSTALL_DIR/examples/moocho/thyra` will not be installed if `-enable-export-makefiles` is not included (or is disabled) or if `-enable-thyra` is missing or `-disable-moocho-thyra` is specified at configure time.

Each installed example contains a simple makefile that is ready to build each of the examples and to demonstrate several important features of MOOCHO. Each makefile shows how to compile and link against the installed header files and libraries. These makefiles use the Trilinos export makefile system to make it easy to get all of the compiler and linker options and get the right libraries in the build process. The user is encouraged to copy these examples to their own directories and modify them to solve their NLPs.

Specific examples are explained below but we first go through the common features of these examples here for one of the `Thyra::ModelEvaluator examples`.

One common feature of all of the installed examples is the makefile that is generated. For the `NLPWBCounterExample` example (that is described in the section Examples of General Serial NLPs with Explicit Jacobian Entries) the makefile looks like:

By using the macros starting with `MOOCHO_` one is guaranteed that the same compiler with the same options are used to build the client's code that were used to build Trilinos. Of particular importance are the macros `MOOCHO_CXX`, `MOOCHO_DEFS`, `MOOCHO_CPPFLAGS`, and `MOOCHO_CXXLD` since these ensure that the same C++ compiler and the same `-D` C/C++ preprocessor definitions are used. These are critical to compiling compatible code in many cases. The macros `MOOCHO_LIBS` contain all of the libraries needed to link executables and they include all of the libraries in their lower-level dependent Trilinos packages. For example, you don't explicitly see the libraries for say Teuchos, but you can be sure that they are there.

This makefile gets created with the following lines commented in or out depending on if `-enable-gnumake` was specified or not when Trilinos was configured:

In the above example, support for GNU Make is enabled which results in scripts being called to clean up the list of include paths and libraries and may have duplicate entries otherwise.

A few different installed NLP examples are described in the following sections.

1.5.2.1 Examples of General Serial NLPs with Explicit Jacobian Entries

- **Waechter and Biegler Counterexample**
  - `NLPInterfacePack::NLPWBCounterExample:` Subclass for a small NLP with n=3 variables and m=2 equality constraints that implements the Waechter and Bielger counterexample [???] which shows global convergence failure from many starting points for many NLP solvers (including many current MOOCHO algorithms).
  - `NLPWBCounterExampleMain.cpp`: Main driver program for solving the NLP.
  - Installed in `$TRILINOS_INSTALL_DIR/example/moocho/NLPWBCounterExample`
- **Scalable Banded Equality and/or Inequality Constrained Example**
  - `NLPInterfacePack::ExampleNLPBanded:` Scalable NLP with `nD` dependent variables, `nI` independent variables, `mI` general inequality constraints, and with a Jacobian with bandwidth of `bw`. This NLP can also be configured to represent a square simulation-only problem (i.e. `nI=0`) and an unconstrained optimization problem (i.e. `nD=0`).
  - `ExampleNLPBandedMain.cpp`: Main driver program for solving the NLP.
  - Installed in `$TRILINOS_INSTALL_DIR/example/moocho/ExampleNLPBanded`

1.5.2.2 Examples of Simulation-Constrained NLPs using Thyra The below examples show subclasses of `EpetraExt::ModelEvaluator` that are used along with `Thyra::EpetraModelEvaluator` and the stratimikos solvers accessed through `Thyra::DefaultRealLinearSolverBuilder`.

- **Simple 4 x 2 serial optimization problem demonstrating the `EpetraExt::ModelEvaluator` interface**
  - `EpetraModelEval4DOpt:` Subclass for a small serial model with n=4 variables and m=2 equality constraints. The purpose of this model is to show the most basic parts of a concrete implementation. This example is only serial however and does not address parallelization issues.

  - `NLPThyraEpetraModelEval4DOptMain.cpp`: Main driver program for solving the NLP.
  - Installed in `$TRILINOS_INSTALL_DIR/example/moocho/thyra/NLPThyraEpetraModelEval4DOpt`

- **Scalable Parallel 2D diffusion/reaction boundary inversion problem**

  - `GLpApp::AdvDiffReactOptModel`: Implements an inversion problem based on a finite-element discretization of a 2D reaction/diffusion state equation. This example shows a more advanced model what includes parallelization in the state space.
  - `NLPThyraEpetraAdvDiffReactOptMain.cpp`: Main driver program for solving the NLP.
  - Installed in `$TRILINOS_INSTALL_DIR/example/moocho/thyra/NLPThyraEpetraAdvDiffReactOpt`

Note: The above examples will only be installed if the configure options `-enable-moocho`, `-enable-thyra`, `-enable-stratimikos`, and `-enable-epetraext-thyra` are all included.

1.5.3 Running MOOCHO to Solve Optimization Problems

Once an NLP is defined and a driver program is in place (see the above driver programs), MOOCHO can then be run to try to solve the optimization problem. When solving an NLP based on `NLPInterfacePack::NLPSerialPreprocessExplJac` one should directly use the solver class `MoochoPack::MoochoSolver`. However, when using an NLP based on `Thyra::ModelEvaluator`, then the solver class `MoochoPack::ThyraModelEvaluatorSolver` should be used. The use of these two classes are demonstrated in the files ??? and ??? respectively.

ToDo: Finish this section!

1.6 Representing Nonlinear Programs for MOOCHO to Solve

As described above, there are two well supported tracts to developing concrete NLP subclasses to be used with MOOCHO. The first type are general NLPs with explicit derivative components that can only be solved in serial. The second type are simulation-constrained NLPs that can be solved on massively parallel computers by utilizing preconditioned iterative linear solvers.

1.6.1 Representing General Serial NLPs with Explicit Jacobian Entries

Warning! This NLP interface is going to most likely change before the next major release of Trilinos. Therefore, it is recommended that users derive their NLPs from the Thyra-based simulation-constrained interfaces described in the next section Representing Simulation-Constrained Parallel NLPs through Thyra.

One type of NLP that MOOCHO can solve are general NLPs where there are explicit gradient and Jacobian entries available. This means that the gradient of the objective function ∇f must be available in vector coefficient form and the gradient of the constraints matrix ∇c (i.e. the rectangular Jacobian $\frac{\partial c}{\partial x} = \nabla c^T$) must be available in sparse matrix form. In this type of problem, a basis matrix for the constraints need not be known *a priori* but this requires the availability of a linear direct solver that can be used to find a square nonsingular basis from a rectangular matrix. There are a few direct solvers available that could in principle find a square basis given a rectangular input matrix but MOOCHO only currently contains wrappers for LAPACK (i.e. dense factorization using `DEGETRF(...)`) and the Harwell Subroutine Library (HSL) routine MA28. The MA28 routine is the only viable option currently supported for large sparse linear systems. In the past, other direct solvers have been experimented with and an ambitious user can provide support for any direct solver they would like (with the ability to find a square basis) by providing an implementation of the `AbstractLinAlgPack::DirectSparseSolver` interface. If your NLP can also provide explicit objective function gradients, then concrete subclasses should derive from the `NLPInterfacePack::NLPSerialPreprocessExplJac` subclass.

ToDo: Copy and paste in writeup from old draft of the MOOCHO user's guide.

1.6.2 Representing Simulation-Constrained Parallel NLPs through Thyra

Another type of NLP that can be solved using MOOCHO are simulation-constrained NLPs where the basis section is known up front. For these types of NLPs, it is recommended that the NLP be specified through the `Thyra::ModelEvaluator` interface and this provides access to a significant linear solver capability through Trilinos. These types of NLPs can be solved in single program multiple data (SPMD) mode in parallel on a massively parallel computer.

The `Thyra::ModelEvaluator` interface uses a different notation than the standard MOOCHO NLP notation. The model evaluator notation is:

$$\begin{array}{ll} \text{minimize} & g(x,p) \\ \text{subject to} & f(x,p) = 0 \\ & x_L \le x \le x_U \\ & p_L \le p \le p_U \end{array}$$

where $x \in \Re^{n_x}$ are the state variables, $p \in \Re^{n_p}$ are the optimization parameters and $f(x,p) = 0$ are the discrete nonlinear state simulation equations. Here the state

Jacobian $\frac{\partial f}{\partial x}$ must be square and nonsingular. The partitioning of variables into state variables x and optimization variables p must be known *a priori* and this partitioning can not change during a solve.

Comparing the MOOCHO notation for optimization problems using variable decomposition methods which is

$$\begin{array}{ll} \text{minimize} & f(x_D, x_I) \\ \text{subject to} & c(x_D, x_I) = 0 \\ & x_{D,L} \leq x_D \leq x_{D,U} \\ & x_{I,L} \leq x_I \leq x_{I,U} \end{array}$$

we can see the mapping between the MOOCHO notation and the `Thyra::ModelEvaluator` notation as summarized in the following table:

It is unfortunate that the notation used with the Model Evaluator interfaces and software are different than those used by MOOCHO. The reason for this change in notation is that the Model Evaluator had to first appeal to the forward solve community where $f(x, p) = 0$ is the standard notation for the parameterized state equation and changing the notation of all of MOOCHO after the fact to match this would be very tedious to perform. We can only hope that the user can keep the above mapping of notation straight between MOOCHO and the Model Evaluator.

Currently, and more so in the near future, a great deal of capability will be automatically available when a user provides an implementation of the `EpetraExt::ModelEvaluator` interface (as shown in the section Examples of Simulation-Constrained NLPs using Thyra). For these types of NLPs, a great deal of linear solver capability is available through the linear solver and preconditioners wrappers in the `Stratimikos` package. In addition, the application will also have access to many other nonlinear algorithms provided in Trilinos (see the Trilinos packages NOX, LOCA, and Rythmos).

1.7 Other Trilinos Packages on which MOOCHO Directly Depends

MOOCHO has direct dependencies on the following Trilinos packages:

- `teuchos`: This package supplies basic utility classes such as `Teuchos::RefCountPtr` and `Teuchos::BLAS` that MOOCHO software is dependent on.
- `rtop`: This package supplies the basic interfaces for vector reduction/transformation operators as well as support code and a library of pre-written RTOp subclasses. Much of the software in MOOCHO depends on this code.

MOOCHO also optionally directly depends on the following Trilinos packages:

- `thyra`: This package supplies interfaces and support software for SPMD and other types of computing platforms and defines the interface `Thyra::ModelEvaluator` for simulation-constrained optimization that MOOCHO can use to define NLPs. See the option `-enable-moocho-thyra` described in the section Configuration of the MOOCHO Package.
- `epetraext`: This package provides an Epetra-specific interface for the model evaluator called `EpetraExt::ModelEvaluator` and contains some concrete examples that are used by MOOCHO.
- `stratimikos`: This package supplies Thyra-based wrappers for several serial direct and massively parallel iterative linear solvers and preconditioners.

1.8 Configuration of the MOOCHO Package

The MOOCHO package's `configure` script (which should be called from the base Trilinos-level configure script) responds to a number of options that affect the code that is built and what code is installed.

Some of the more important configuration options are:

- `-enable-moocho`: Causes the MOOCHO package and all of its dependent packages to be enabled and built. Without this option, there will be no MOOCHO header files or libraries included in the installation of Trilinos (i.e. using `make install`).
- `-enable-moocho-ma28`: Causes the MOOCHO package to compile in support for the sparse solver HSL MA28. Currently, this is the only supported direct solver for large sparse systems where the basis matrix is not know up front.
- `-enable-moocho-thyra`: Causes the MOOCHO package to compile in support for `Thyra::ModelEvaluator` to support massively parallel simulation-constrained optimization. Note that this option will be turned on by default if `-enable-moocho` and `-enable-thyra` are both included.
- `-enable-moocho-stratimikos`: Causes the examples in the MOOCHO package to compile in support for the linear solver wrappers through the Stratimikos package. None of the software in the MOOCHO library has any dependence on Stratimikos but none of the examples that Thyra depend on it will be compiled or installed if this option is not included. Note that this option will be turned on by default if `-enable-moocho` and `-enable-stratimikos` are both included. This option is only meaningful if Thyra support is enabled.

- `-enable-export-makefiles`: Causes the installation of the MOOCHO package (an other Trilinos packages) to have the makefile fragments `Makefile.export.moocho` and `Makefile.export.moocho.macros` installed in the installation directory `$TRILINOS_INSTALL_DIR/include` for use by external makefiles (see the section Examples of General Serial NLPs with Explicit Jacobian Entries). This option also causes the examples described in the section Examples of Simulation-Constrained NLPs using Thyra to be installed.

See the output from `Trilinos/pacakges/moocho/configure -help` for a complete listing of all of the configure options for which MOOCHO responds.

The MOOCHO package is also affected by configure options passed to other packages. Here are some of of these options:

- `-enable-teuchos-debug`: Causes a great deal of error checking code to be added to MOOCHO software.
- `-enable-thyra`: Enables all Thyra-based software by default and enables the MOOCHO/Thyra adapters by default.
- `-enable-epetraext-thyra`: Causes the examples that depend on `EpetraExt::ModelEvaluator` described in the section Examples of Simulation-Constrained NLPs using Thyra to be compiled and installed. Note that this is enabled by default if `-enable-thyra` and `-enable-epetraext` are both included.
- `-enable-stratimikos`: Enables support for Stratimikos. Note that this automatically enables `-enable-moocho-stratimikos` by default.
- `-enable-amesos`: Enables support for the Amesos linear solvers that can be access through Stratimikos.
- `-enable-aztecoo`: Enables support for the AztecOO linear solvers that can be access through Stratimikos.
- `-enable-belos`: Enables support for the Belos linear solvers that can be access through Stratimikos.
- `-enable-ifpack`: Enables support for the Ifpack preconditioners that can be access through Stratimikos.
- `-enable-ml`: Enables support for the ML preconditioners that can be access through Stratimikos.

Note that the above options will not be listed by `Trilinos/packages/moocho/configure -help` but instead are listed by `Trilinos/configure -help=recursive`.

1.9 Individual MOOCHO doxygen collections

Below are links to individual doxygen collections that make up MOOCHO:

- MoochoUtilities: Collection of a small amount of utility code that is peculiar to MOOCHO. Some of the software that is now in Teuchos such as Teuchos::RefCountPtr and Teuchos::CommandLineProcessor where once in this collection.
- IterationPack: Framework for building iterative algorithms that MOOCHO is based on.
- RTOpPack: Legacy RTOp code that predates Thyra the Trilinos RTOp package but it still used by MOOCHO. The current version of the Trilinos RTOp package was developed from refactored code that once lived in this collection.
- DenseLinAlgPack: A C++ class library for dense, BLAS-compatible, serial linear algebra that is similar to classes like Teuchos::SerialDenseVector and Teuchos::SerialDenseMatrix. This class library is used exclusively by MOOCHO to deal with serial dense linear algebra.
- AbstractLinAlgPack: A C++ class library for abstract linear algebra. These interfaces predate and helped to inspire Thyra but at this point should be considered legacy software that should only be used within MOOCHO. It is likely that a future refactoring of MOOCHO will involve largly removing these classes and using Thyra directly instead.
- NLPInterfacePack: Set of abstract interfaces based on AbstractLinAlgPack for representing nonlinear programs (NLPs) (i.e. optimization problems). These interfaces serve a similar role as the Thyra::ModelEvaluator interface but there are many differences. In the future, it is likely that these interfaces will be refactored to look more like the Thyra::ModelEvaluator interface but are likely to remain distinct.
- ConstrainedOptPack: Collection of utility software for building constrained optimization algorithms that is based on AbstractLinAlgPack. Included here are interfaces and adapters for QP solvers (with QPSchur being included by default), line search interfaces and implementations, range/null space decompositions and other such capabilities.
- MoochoPack: Provides nonlinear optimization algorithms for primarily rSQP methods based on the IterationPack framework. This is where the real algorithmic meat of nonlinear programing is found in MOOCHO. This collection provides the "Facade" MoochoPack::MoochoSolver.
- MOOCHO/Thyra Adapters: Provides adapter classes for allowing MOOCHO to solve simulation-constrained optimization problems presented as

`Thyra::ModelEvaluator` objects. Also included is the higher-level "Facade" class `MoochoPack::ThyraModelEvaluatorSolver`.

1.10 Browse all of MOOCHO as a single doxygen collection

You can browse all of MOOCHO as a `single doxygen collection`. Warning, this is not the recommended way to learn about MOOCHO software. However, this is a good way to browse the `directory structure of MOOCHO`, to `locate files`, etc.

1.11 Links to other documentation collections

- `Thyra`: This package defines basic interfaces and support software for abstract numerical algorithms.
- `Thyra ANA Operator/Vector Adapters for Epetra`: This software includes the basic adapters needed to wrap Epetra objects and Thyra objects.
- `Various Thyra Adapters for EpetraExt`: Included here are adapters and interfaces that allow a perspective nonlinear application to specify everything needed to define a wide range of nonlinear problems in terms by subclassing an Epetra-based version of the `Thyra::ModelEvaluator` interface (called `EpetraExt::ModelEvaluator`). This software allows an appropriately defined Epetra-based model to be used to define a Thyra-based model to be used to define an optimization problem that MOOCHO can then solve.
- `Stratimikos: Unified Wrappers for Thyra Linear Solver and Preconditioner Adapters`: Stratimikos contains neatly packaged access to all of the Thyra linear solver and preconditioner wrappers. Currently, these allow the creation of linear solvers for nearly any `Epetra_RowMatrix` object.

MOOCHO Notation	`Thyra::ModelEvaluator` **Notation**	`Thyra::ModelEvaluator` **Description**
m	n_x	Number of state variables
$n-m$	n_p	Number of optimization parameters
n	$n_x + n_p$	Total number of optimization variables
$x_D \in \Re^m$	$x \in \Re^{n_x}$	State variables
$x_I \in \Re^{n-m}$	$p \in \Re^{n_p}$	Optimization parameters
$c(x_D, x_I) \in \Re^n \rightarrow \Re^m$	$f(x,p)\Re^{n_x+n_p} \rightarrow \Re^{n_x}$	State equation residual function
$f(x_D, x_I) \in \Re^n \rightarrow \Re$	$g(x,p)\Re^{n_x+n_p} \rightarrow \Re$	Objective function
$C \in \Re^{m \times m}$	$\frac{\partial f}{\partial x} \in \Re^{n_x \times n_x}$	Nonsingular state Jacobian
$N \in \Re^{m \times n-m}$	$\frac{\partial f}{\partial p} \in \Re^{n_x \times n_p}$	Optimization Jacobian
$\nabla_D f^T \in \Re^{1 \times m}$	$\frac{\partial g}{\partial x} \in \Re^{1 \times n_x}$	Derivative of objective with respect to state variables
$\nabla_I f^T \in \Re^{1 \times n-m}$	$\frac{\partial g}{\partial p} \in \Re^{1 \times n_p}$	Derivative of objective with respect to optimization parameters

Table 1: Mapping of notation between MOOCHO and `Thyra::ModelEvaluator` for simulation-constrained optimization problems.